

PEMS08 交通流预测系统

基于图卷积网络(GCN)特征提取和机器学习的多步交通流预测系统，支持多种预测模型和交互式可视化。

项目概述

本项目使用PEMS-08数据集（170个交通传感器节点）进行交通流预测，采用"GCN全局特征提取 + 机器学习节点级预测"的创新架构，并提供专业的交互式可视化界面。

核心特性

- 多模型支持：BP神经网络、SVM、KNN、线性回归及集成方法
- 图神经网络：使用GCN提取节点间空间关系特征
- 交互式可视化：基于Dash的现代化Web界面
- 一键运行：训练完成自动启动可视化应用
- 多步预测：支持12步时间序列预测

项目架构

```
traffic-prediction/
├── data/                                # 数据目录
│   ├── PEMS-08/
│   │   ├── pems08.npz                 # 主数据文件
│   │   └── distance.csv               # 节点距离矩阵
│   └── models/                       # 模型定义
│       ├── base.py                   # 基础模型类
│       ├── gcn.py                    # 图卷积网络
│       └── ml_models/                # 机器学习模型
│           ├── bp.py                 # BP神经网络
│           ├── svm.py                 # 支持向量机
│           ├── knn.py                # K近邻
│           ├── linear.py              # 线性回归
│           └── ensemble/              # 集成学习
│               ├── bagging.py         # Bagging集成
│               ├── adaboost.py        # AdaBoost集成
│               └── stacking.py        # Stacking集成
├── utils/                             # 工具模块
│   ├── data_processor.py             # 数据处理
│   ├── metrics.py                    # 评估指标
│   └── visualization.py              # 可视化工具
├── app/                               # Web可视化应用
│   ├── app.py                        # Dash应用主文件
│   └── data/                         # 可视化数据
│       ├── node_coords.csv           # 节点坐标
│       ├── predictions.npy           # 预测结果
│       └── y_true.npy                 # 真实值
└── train.py                           # 训练主程序
```

快速开始

环境要求

- Python 3.8+
- PyTorch 1.9+
- PyTorch Geometric
- scikit-learn
- Dash & Plotly
- NumPy & Pandas

安装依赖

```
# 创建虚拟环境
conda create -n traffic-pred python=3.9
conda activate traffic-pred

# 安装PyTorch
conda install pytorch torchvision torchaudio -c pytorch

# 安装PyTorch Geometric
pip install torch-geometric

# 安装其他依赖
pip install scikit-learn dash plotly pandas numpy networkx
```

一键运行

```
# 训练模型并自动启动可视化
python train.py --model bp

# 浏览器访问
# http://127.0.0.1:8050
```

使用指南

1. 模型训练

支持多种预测模型：

```
# BP神经网络（推荐）
python train.py --model bp

# AdaBoost集成（效果最佳）
```

```
python train.py --model adaboost

# Bagging集成
python train.py --model bagging

# Stacking集成
python train.py --model stacking

# 其他单一模型
python train.py --model svm
python train.py --model knn
python train.py --model linear
```

2. 自定义参数

```
# 自定义模型参数
python train.py --model bp --model_params '{"hidden_sizes": [128, 64], "learning_rate": 0.01}'




# 自定义数据参数
python train.py --model bp --seq_len 24 --pred_len 6
```

3. 可视化界面

训练完成后自动启动，或手动启动：

```
cd app
python app.py
```

界面功能：

-  节点分布图：显示170个传感器节点的网络拓扑
-  预测曲线：点击节点查看第12步预测vs真值对比
-  实时信息：显示节点坐标、预测步长等信息

技术架构

数据流程

原始数据 → 时间序列切片 → GCN特征提取 → 节点级预测 → 结果可视化

核心算法

1. GCN特征提取

- 输入：(batch, nodes, features)
- 输出：(batch, nodes, gcn_dim)

- 提取节点间空间关系特征

2. 节点级预测

- 输入: `(batch*nodes, gcn_dim)`
- 输出: `(batch*nodes, pred_len)`
- 每个节点独立预测12步

3. 反归一化

- 只对目标特征（交通流量）进行反归一化
- 保持预测结果的真实尺度

算法详细实现

1. 数据预处理算法

时间序列切片

```
def create_sequences(data, seq_len, pred_len):  
    """  
    将原始时间序列数据切片为训练样本  
  
    输入: data.shape = (T, N, F) # T=时间步, N=节点数, F=特征数  
    输出: X.shape = (samples, seq_len, N, F)  
           Y.shape = (samples, N, pred_len)  
    """  
    X, Y = [], []  
    for i in range(len(data) - seq_len - pred_len + 1):  
        # 输入序列: 过去12个时间步的所有特征  
        x = data[i:i+seq_len] # (12, 170, 3)  
        # 目标序列: 未来12个时间步的流量特征  
        y = data[i+seq_len:i+seq_len+pred_len, :, 0] # (12, 170)  
        X.append(x)  
        Y.append(y.T) # 转置为 (170, 12)  
    return np.array(X), np.array(Y)
```

数据归一化

```
def normalize_data(data):  
    """  
    使用z-score标准化  
     $X_{norm} = (X - \mu) / \sigma$   
    """  
    mean = np.mean(data, axis=(0, 1), keepdims=True)  
    std = np.std(data, axis=(0, 1), keepdims=True)  
    normalized = (data - mean) / (std + 1e-8)  
    return normalized, mean, std
```

2. 图卷积网络(GCN)实现

网络架构

```
class GCN(nn.Module):
    def __init__(self, input_dim=3, hidden_dim=64, output_dim=32, num_layers=2):
        super().__init__()
        self.layers = nn.ModuleList()

        # 第一层: input_dim -> hidden_dim
        self.layers.append(GCNConv(input_dim, hidden_dim))

        # 中间层: hidden_dim -> hidden_dim
        for _ in range(num_layers - 2):
            self.layers.append(GCNConv(hidden_dim, hidden_dim))

        # 输出层: hidden_dim -> output_dim
        self.layers.append(GCNConv(hidden_dim, output_dim))

        self.dropout = nn.Dropout(0.2)
        self.activation = nn.ReLU()
```

GCN前向传播

```
def forward(self, x, edge_index):
    """
    GCN前向传播过程

    输入: x.shape = (batch_size, num_nodes, input_features)
           edge_index.shape = (2, num_edges)
    输出: x.shape = (batch_size, num_nodes, output_features)
    """
    batch_size, num_nodes, _ = x.shape

    # 重塑为 (batch_size * num_nodes, features)
    x = x.view(-1, x.size(-1))

    # 扩展边索引以处理批次数据
    edge_indices = []
    for i in range(batch_size):
        edge_indices.append(edge_index + i * num_nodes)
    batch_edge_index = torch.cat(edge_indices, dim=1)

    # 逐层传播
    for i, layer in enumerate(self.layers):
        x = layer(x, batch_edge_index)
        if i < len(self.layers) - 1:
            x = self.activation(x)
            x = self.dropout(x)

    # 重塑回 (batch_size, num_nodes, output_features)
    return x.view(batch_size, num_nodes, -1)
```

图构建算法

```
def build_graph_from_distance(distance_matrix, threshold=0.1):  
    """  
    基于距离矩阵构建图的邻接关系  
  
    使用高斯核函数:  $w_{ij} = \exp(-d_{ij}^2/\sigma^2)$   
    """  
    # 计算高斯权重  
    sigma = np.std(distance_matrix)  
    weights = np.exp(-distance_matrix**2 / sigma**2)  
  
    # 设置阈值, 过滤弱连接  
    adjacency = (weights > threshold).astype(float)  
  
    # 转换为边索引格式  
    edge_index = np.array(np.nonzero(adjacency))  
    return torch.LongTensor(edge_index)
```

3. 机器学习模型实现

BP神经网络

```
class MultiOutputBP(BaseModel):  
    def __init__(self, input_size, hidden_sizes=[64, 32], learning_rate=0.001):  
        super().__init__('MultiOutputBP')  
        self.network = self._build_network(input_size, hidden_sizes, 12)  
        self.optimizer = torch.optim.Adam(self.network.parameters(), lr=learning_rate)  
        self.criterion = nn.MSELoss()  
  
    def _build_network(self, input_size, hidden_sizes, output_size):  
        """构建多层感知机网络"""  
        layers = []  
        prev_size = input_size  
  
        for hidden_size in hidden_sizes:  
            layers.extend([  
                nn.Linear(prev_size, hidden_size),  
                nn.ReLU(),  
                nn.Dropout(0.2)  
            ])  
            prev_size = hidden_size  
  
        layers.append(nn.Linear(prev_size, output_size))  
        return nn.Sequential(*layers)  
  
    def fit(self, X, y, epochs=100, batch_size=32):  
        """训练过程"""  
        dataset = TensorDataset(torch.FloatTensor(X), torch.FloatTensor(y))  
        dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

```

self.network.train()
for epoch in range(epochs):
    total_loss = 0
    for batch_X, batch_y in dataloader:
        self.optimizer.zero_grad()

        # 前向传播
        predictions = self.network(batch_X)
        loss = self.criterion(predictions, batch_y)

        # 反向传播
        loss.backward()
        self.optimizer.step()

    total_loss += loss.item()

    if epoch % 20 == 0:
        print(f'Epoch {epoch}, Loss: {total_loss/len(dataloader):.4f}')

```

集成学习算法

AdaBoost实现

```

def fit(self, X, y):
    """AdaBoost训练算法"""
    n_samples = len(X)
    # 初始化样本权重
    sample_weights = np.ones(n_samples) / n_samples

    for i in range(self.n_estimators):
        # 训练基模型
        model = self.base_model_class(**self.base_params)
        model.fit(X, y, sample_weight=sample_weights)

        # 计算预测误差
        predictions = model.predict(X)
        errors = np.abs(predictions - y)

        # 多输出处理：对所有输出维度求平均
        if errors.ndim > 1:
            errors = np.mean(errors, axis=1)

        # 计算加权误差率
        weighted_error = np.sum(sample_weights * errors) / np.sum(sample_weights)

        # 早停条件
        if weighted_error >= 0.5:
            break

        # 计算模型权重
        alpha = self.learning_rate * 0.5 * np.log((1 - weighted_error) / weighted_error)

```

```

# 更新样本权重
sample_weights *= np.exp(alpha * errors)
sample_weights /= np.sum(sample_weights) # 归一化

self.models.append(model)
self.weights.append(alpha)

```

Stacking实现

```

def fit(self, X, y):
    """Stacking两层训练"""
    # 第一层：训练基模型
    meta_features = []
    for model_class in self.base_models:
        model = model_class()
        model.fit(X, y)

        # 生成元特征（交叉验证预测）
        predictions = model.predict(X)
        meta_features.append(predictions)
        self.trained_base_models.append(model)

    # 拼接元特征
    meta_X = np.column_stack(meta_features) # (samples, n_models * pred_len)

    # 第二层：训练元模型
    self.trained_meta_model = self.meta_model(**self.meta_params)
    self.trained_meta_model.fit(meta_X, y)

```

4. 训练流程算法

完整训练Pipeline

```

def train_pipeline(data_path, model_name, seq_len=12, pred_len=12):
    """完整的训练流程"""

    # 1. 数据加载与预处理
    processor = DataProcessor(data_path, seq_len, pred_len)
    (X_train, Y_train), (X_val, Y_val), (X_test, Y_test), adj = processor.prepare_data()

    # 2. 构建图结构
    edge_index = adj_to_edge_index(adj)

    # 3. GCN特征提取
    gcn_features_train = processor.extract_gcn_features(X_train[:, -1, :, :], edge_index)
    gcn_features_test = processor.extract_gcn_features(X_test[:, -1, :, :], edge_index)

    # 4. 数据重塑：节点级预测
    n_samples, n_nodes, gcn_dim = gcn_features_train.shape
    X_train_flat = gcn_features_train.reshape(n_samples * n_nodes, gcn_dim)
    Y_train_flat = Y_train.reshape(n_samples * n_nodes, pred_len)

```



```

# 5. 模型训练
model = get_model(model_name, input_size=gcn_dim)
model.fit(X_train_flat, Y_train_flat)

# 6. 预测与评估
predictions = model.predict(X_test_flat)
predictions = predictions.reshape(n_test_samples, n_nodes, pred_len)

# 7. 反归一化
predictions = processor.inverse_normalize_target(predictions)
y_true = processor.inverse_normalize_target(Y_test)

return predictions, y_true

```

5. 评估指标算法

多维度评估

```

def calculate_comprehensive_metrics(y_true, y_pred):
    """计算多维度评估指标"""

    # 整体指标
    mae = np.mean(np.abs(y_true - y_pred))
    mse = np.mean((y_true - y_pred) ** 2)
    rmse = np.sqrt(mse)
    mape = np.mean(np.abs((y_true - y_pred) / (y_true + 1e-8))) * 100

    # 节点级指标
    node_metrics = {}
    for node in range(y_true.shape[1]):
        node_mae = np.mean(np.abs(y_true[:, node, :] - y_pred[:, node, :]))
        node_metrics[f'node_{node}_mae'] = node_mae

    # 时间步级指标
    horizon_metrics = {}
    for step in range(y_true.shape[2]):
        step_mae = np.mean(np.abs(y_true[:, :, step] - y_pred[:, :, step]))
        horizon_metrics[f'step_{step+1}_mae'] = step_mae

    return {
        'overall': {'MAE': mae, 'MSE': mse, 'RMSE': rmse, 'MAPE': mape},
        'by_node': node_metrics,
        'by_horizon': horizon_metrics
    }

```

6. 算法优化策略

内存优化

- 批次处理：限制训练样本数量为2000，避免内存溢出

- **梯度累积**：对于大批次数据使用梯度累积技术
- **特征缓存**：缓存GCN提取的特征，避免重复计算

计算优化

- **并行计算**：利用PyTorch的并行计算能力
- **GPU加速**：自动检测并使用CUDA加速
- **早停机制**：在验证集上监控性能，避免过拟合

数值稳定性

- **梯度裁剪**：防止梯度爆炸
- **权重初始化**：使用Xavier初始化
- **批归一化**：在深层网络中使用BatchNorm

模型性能

模型	MAE	MSE	RMSE	特点
AdaBoost	87.19	20723.02	143.95	🏆 效果最佳
Bagging	89.34	17602.97	132.68	🥈 稳定性好
BP神经网络	97.92	21297.35	145.94	🥉 收敛快速
Stacking	100.60	23680.56	153.88	🏆 复杂度高

配置说明

数据配置

- **序列长度**： `seq_len=12` （输入12个时间步）
- **预测长度**： `pred_len=12` （预测12个时间步）
- **节点数量**： 170个交通传感器
- **特征维度**： 3（流量、占有率、速度）

模型配置

- **GCN维度**： 32维特征向量
- **训练样本**： 限制2000个（加速调试）
- **设备支持**： 自动检测CUDA/CPU

数据说明

PEMS-08数据集

- 来源：加州交通管理系统
- 时间范围：2016年7月-8月
- 采样频率：5分钟间隔
- 节点数量：170个检测器
- 数据格式：(时间步, 节点, 特征)

可视化特性

界面设计

- 现代化UI：卡片式布局，蓝灰配色
- 响应式设计：左右分栏，48%宽度布局
- 交互体验：点击节点即时显示预测曲线
- 状态提示：自动识别真实/模拟数据

图表功能

- 节点分布：基于距离关系的网络拓扑可视化
- 预测对比：真值vs预测值曲线对比
- 悬停信息：详细的数值和坐标信息

开发指南

添加新模型

1. 在 `models/ml_models/` 创建新模型文件
2. 继承 `BaseModel` 类
3. 实现 `fit` 和 `predict` 方法
4. 在 `train.py` 中注册模型

自定义评估指标

在 `utils/metrics.py` 中添加新的评估函数：

```
def custom_metric(y_true, y_pred):  
    # 自定义指标计算  
    return metric_value
```

常见问题

Q: 训练时显示CUDA out of memory?

A: 减少 `max_train` 参数或使用CPU训练

Q: 可视化显示模拟数据?

A: 确保先运行 `train.py` 生成预测数据

Q: 集成模型训练很慢?

A: 减少 `n_estimators` 参数或使用更简单的基模型



许可证

MIT License



贡献

欢迎提交Issue和Pull Request!

★ 如果这个项目对你有帮助, 请给个Star!